From Pixels to Plays: Developing an Image-Based Agent for SuperTuxKart Ice-Hockey

Ajay Rathore, Kostiantyn Shevel, Byron King

The University of Texas at Austin, Departments of Computer and Data Science

ABSTRACT: This paper presents the development and performance analysis of an image-based agent for a SuperTuxKart ice-hockey player. The task involves programming an autonomous agent to participate in 2 vs 2 SuperTuxKart ice-hockey tournaments. The primary objective is to score as many goals as possible to win matches against various agents. The controller for the image-based agent, implemented in image_agent/player.py, relies on the visual input from each player's perspective to infer critical game elements such as the puck's location and the agent's position. This approach requires the integration of a vision-based model for object detection and spatial reasoning, coupled with a hand-tuned controller to navigate and strategize within the game environment. Our strategy was to generate a training dataset of images from games played locally against various AI agents. Our final, best-performing implementation featured two models used in the controller to navigate the kart. We first adapted a pre-trained fully convolutional neural network model from HW4 trained to detect object centers on a dense heatmap and used this model to guide the kart for the [redacted] We then trained another FCN model used in the controller for [redacted] which featured a novel tactic used in training data generation that captured [redacted] Our agent's performance was ultimately successful. The final model we trained achieved a 0.9787 validation accuracy and average validation loss of 0.0213 for puck object detection after 50 epochs. Ultimately, our agent managed to score 4 goals scored in 4 games against the Geoffrey agent, 2 goals scored in 4 games against the Jurgen agent, 4 goals scored in 4 games against the Yann agent, and 2 goals scored in 4 games against the Yoshua agent. Our agent's success underscores the promise of current computer vision techniques and sheds light on the potential of these approaches to shape future advancements in real-world domains like autonomous transportation.

INTRODUCTION

This paper details the development of an image-based agent designed for the SuperTuxKart ice-hockey game. This task involves programming an autonomous agent capable of competing in 2 vs 2 matches with the primary goal of scoring goals against AI opponents. The image-based agent developed in image_agent/player.py relies on visual input frames retrieved during gameplay, but without direct access to the state of the puck or opposing players during the game. We chose to implement an image-based agent instead of a state-based agent in order to situate our research in the broader canon of computer vision (instead of the field of reinforcement learning) and to draw parallels between our implementation and the important real-world problem of autonomous driving.

While playing the game the image-based agent receives continuous image frames that represent the state of the game from the agent's kart's perspective. The image frames are processed to extract important game components such as the location of the agent. This processing introduced the problem of coordinate translation, which is one of the most challenging parts of the project. Coordinate translation is necessary, for example, after identification of the puck's position in the frame. It includes translating from image to camera to world coordinates followed by a series of coordinate transformations to infer the puck's location in the game's world coordinate system after converting the puck's coordinates from the 2D image plane to the 3D world space. When translating from image to world coordinates, this process requires normalizing the image coordinates back so that they are in the same frame of reference as the camera coordinates.

This critical step ensures that the puck's location in the image aligns with its actual position in the game's 3D environment without distorting the location, enabling the agent to make strategic decisions based on accurate spatial information. Nonetheless, we struggled to properly map world coordinates to image coordinates. Instead, we settled on an implementation that does not transform image coordinates to world coordinates directly. Instead, [redacted] The controller is designed to use this information to make decisions such as navigating towards the puck or positioning for defensive or offensive plays. The image-based agent's performance is then governed by a hand-tuned controller.

In the following Methodology section, we outline the details of our image-based agent implementation. There are two key components of the agent: the models and the controller. We ultimately used two models to dictate the movements of our kart. The first is a pre-trained model from HW4, the other we trained specifically to find the image center of the puck so that we could steer our karts toward it. We discuss the architecture of the models we used for object detection as well as the data-generating mechanism and the training process of the model we trained from scratch. In the same section, we explain how the controller works and how we ultimately settled on a two-model approach to designing the controller.

In the subsequent Training Results section, we present the results of the model we trained. We discuss the hyperparameters used in the final model implementation as well as key training statistics.

We also discuss the relevant performance statistics for the trained model and report how the image based agent ultimately performed against its AI opponents in the Canvas grader.

In the Error Analysis section, we go further into the development of our image-based agent. The development of the models and the operation of the kart using the predictions of these models was an iterative, experimental process. We began by tweaking the behavior of just the controller to see how the kart functioned. We then tried a few different model implementations before settling on a variation of the dense heatmap object center detection model used in HW4 to maneuver our two karts to score goals against our opponents. We changed the behavior of our controller both experimentally (by seeing how different changes to the agent behavior and different uses of the model predictions yielded different results) and analytically (by analyzing the errors of our models as we developed them) to arrive at a final successful implementation. In this section, we recount that development process and provide insights into how we arrived at our final controller result.

Finally, in the Conclusion section we summarize the objective and the results and provide ideas for how we could improve on the implementation in the future. Ideas for improvements include different ways of generating training and test data as well as improvements to the controller that improve its effectiveness. We also situate our implementation into the broader landscape of computer vision research and the real-world applications of the techniques we apply in our paper.

METHODOLOGY

Data Generation Mechanism

Our final implementation featured two models, one that was pre-trained and one that was trained using data generated during gameplay. We now describe the data generation method for the latter model.

To create a training set of images, we experimented with generating images of various coordinate systems, sizes, and transformations. In order for the data generation mechanism to produce high-quality training data for our model, we needed to develop three principle innovations to address coordinate mapping, image sizes, and transformations respectively.

The first was to create a function that takes world coordinates, maps them to the equivalent image coordinates, and [redacted] We tested several image sizes (such as the [redacted] formats) before settling on the [redacted] image size for the final model training. We ultimately settled on this size because [redacted]

The second innovation was to rewrite the DataLoader class from HW4 so that it would accep[redacted] Trial and error with the performance of this modified class revealed that th[redacted] was more performant than the [redacted] in the instructor implementation for HW4.

During gameplay, we also experimented with creating training data in the ***act*** method of the ***Team*** class versus in the ***runner.py*** script directly. We noticed that creating [redacted] more effective because [redacted]

Using this data generation technique, we could in one game generate up to 4000 images. The models we trained used images created from between 8-12 entire matches which resulted in roughly 30.000 – 50.000 total images. We used this approach to create images that we used for training and validation. We randomly selected images for training and validation i[redacted] proportion.

The final innovation concerns creating target data for model training. In the data generation process, we tested several image transformation effects on the original image in order to make the puck's location more pronounced to yield better results during training. The final method we settled [redacted]

Figure 1 features [redacted] More examples can be found in Appendix A.

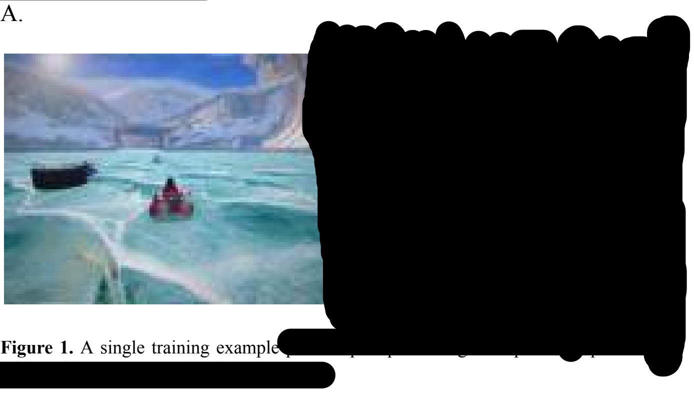


Figure 1. A single training example [redacted]

The data generation mechanism, including the three innovations needed to produce high quality training data, are summarized in the workflow diagram in Figure 2.

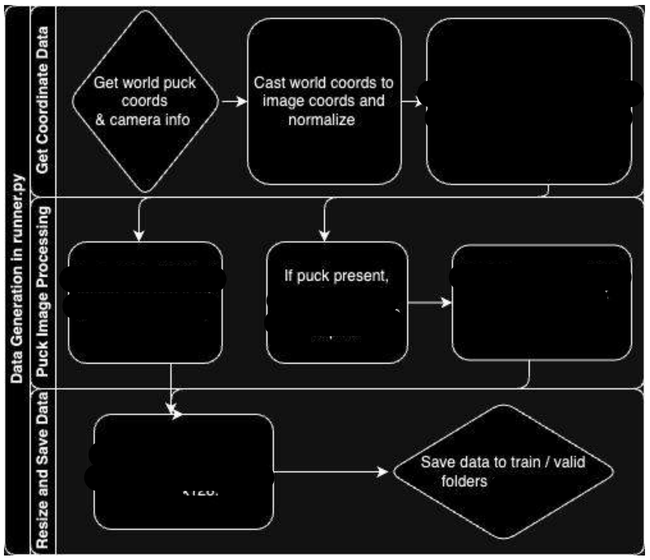

Data Generation in runner.py
Get Coordinate Data
Get world puck coords & camera info
Cast world coords to image coords and normalize
[redacted]
Puck Image Processing
[redacted]
If puck present,
[redacted]
Resize and Save Data
[redacted]
Save data to train / valid folders

Figure 2. The workflow captures the three primary components of the data generation mechanism, namely 1) getting coordinate data 2) processing the image of the puck 3) resizing and saving the processed data.

We now describe the model architecture we selected for the model we trained using this data generation technique.

Model Selection and Architecture

As discussed, we used two models in our implementation (one pre-trained and one trained using data produced by the previously described mechanism). Each of these are fully convolutional networks adopted and heavily modified from the HW3 master solution. The composition of each is similar in that they feature medium width architectures that have 16, 32, 64, 128 layers each with a single skip connection.

The full model architecture for the final implementation of the trained model is shown in the simplified schematic in Figure 3. A more detailed architecture diagram can be found in Appendix B.

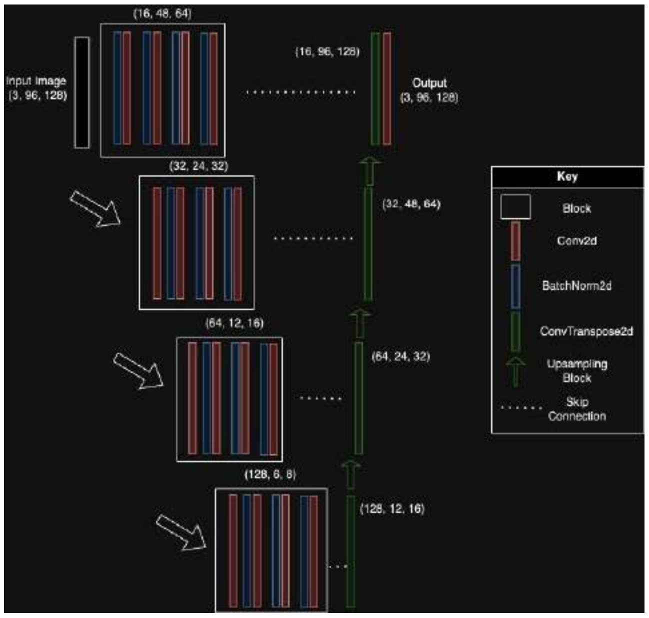


Figure 3. The final trained model used in the controller features a U-Net style architecture with encoder and decoder paths. There are 4 downsampling and upsampling paths each with a single skip connection.

We now describe the agent controller and the design choices we made as well as how the controller uses the trained and pre-trained model predictions.

Agent Controller Design

We designed the controller using a trial-and-error approach. We realized that using different kart types required slightly different controller behavior, so we developed our controller logic just using th[redacted]kart.

Initially we found that the pre-trained model from HW4 for objection detection produced results that were better than the ones achieved by the model we trained on the puck-detection data. However, we soon realized that the pre-trained mode[redacted]We speculate that this is because the algorithm would mistake the interior of the kart (which is black) as the puck, which required us[redacted]
This was likely an error-prone procedure.

Interestingly, when we produced a better model on novel gameplay-generated training data, we found that using the pre-trained model[redacted]produced more goals than using only the trained model. We decided to rely on the pre-trained mode[redacted]

[redacted]he second algorithm is used to steer the kart. In this second phase, the controller operates in a simple fashion.[redacted] dictionary for tracking all of the relevant information about the field so that the player agent could take into account this data. This includes the kart velocity, the angle to the puck, the distance to its own goal, and the direction and location of the opponent's goal. We used this information to steer this kart backwards and forwards[redacted]

We experimented with various ways of combining the two algorithms and switching between the two. The final implementation consists of using the pre-trained model just for the [redacted] To get a sense of the time that elapses from the start of the match, we used a frame counter and [redacted]o aid in understanding how the controller logic differs by frame, we provide in Table 1 pseudocode that describes each kart's behavior.

Frame Number	Kart_0 (Left/Right)	Kart_1 (Central)
[redacted]	[redacted]	[redacted]

Table 1. This table provides a pseudocode description of how the controller's behavior changes after a certain number of frames elapse.

TRAINING RESULTS

When training our model, we tried several loss functions but achieved the best results with th[redacted] function. Our highest performing model also used[redacted] We trained our model for 50 epochs and reported on the average training loss and accuracy while training to ensure that the model was properly converging. We report the hyperparameters of the most performant model in Table 2, as well as its training statistics in Table 3.

Hyperparameter	**Value**
Epochs	50
Learning Rate	[redacted]
Loss Function	[redacted]
Batch Size	[redacted]
Weight Decay	[redacted]

Table 2. This table contains the final hyperparameters for in the trained model used by the controller in the final implementation.

Training Statistic	**Value**
Training Set Size	33,333 images
Validation Set Size	16,666 images
State Dictionary Size	3.1 mb
Mean Training Loss After 50 Epochs	0.0206
Mean Training Accuracy After 50 Epochs	0.9794
Mean Validation Loss After 50 Epochs	0.0213
Mean Validation Accuracy After 50 Epochs	0.9787

Table 3. This table contains the final training and validation statistics for the trained model used by the controller in the final implementation. We report only the statistics after the 50th training epoch.

We also display the average validation accuracy and loss for each epoch during training in the figures below. A tabular view of these statistics is available in Appendix C. It is apparent that after 25 epochs the accuracy plateaus and the next 25 epochs the improvement is only marginal.

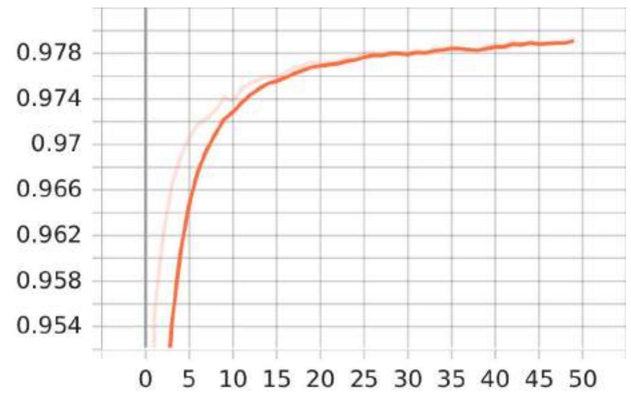


Figure 4. This chart plots over 50 epochs the average validation set accuracy. The steepest improvement is within the first 25 epochs, while in the subsequent 25 epochs the rate of growth is only marginal.

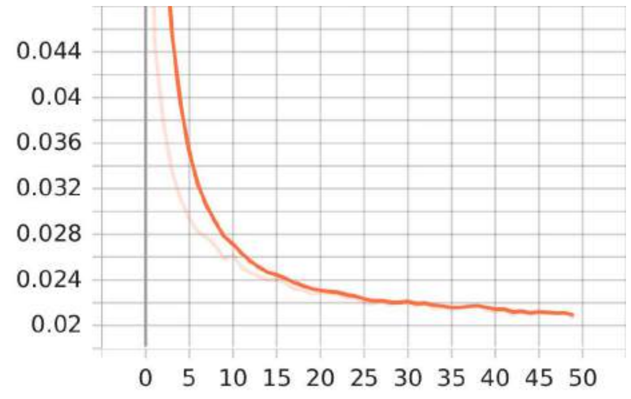


Figure 5. This chart plots over 50 epochs the average validation loss. The inverse relationship is evident — the steepest decrease in loss is within the first 25 epochs.

After training the model with the above hyperparameters, and after tweaking the controller to achieve optimal performance with the dual-model approach, we tested the agent against the local grader and the Canvas grader. Our best Canvas grader performance is summarized in Table 4, and achieved a final grade of 76. Our agent performed the best against the Geoffrey and Yann agents, scoring 4 goals in 4 matches against each. Our agent performed slightly worse against the Jurgen and Yoshua agents, scoring 2 goals in 4 matches played against each agent.

Agent	**Goals Scored**	**Matches Played**
Geoffrey	4	4
Jurgen	2	4
Yann	4	4
Yoshua	2	4

Table 4. This table contains the best Canvas performance of our agent against the opponent agents.

In the next section, we describe the iterative process we used to develop the controller logic and the selection of the models we used in the controller.

ERROR ANALYSIS

During the iterative development of the controller we used to manipulate the behavior of our karts, we encountered several obstacles both in the controller design and in the development of the models that dictated its behavior. We recount our experiments here to give a sense of the process we took in reaching our final implementation.

Model Training Experiments

Our first approach to data generation was to generate images using ***runner.py***, but to have the ground truth based on converting world coordinates to screen coordinates. We then combined both the images with and without the puck in a single training set and then trained a U-Net model on this dataset to predict the puck's screen location. However, this approach yielded unsatisfactory training performance statistics, and the agent was unable to score any goals against the grader opponents.

We also ran other experiments using a two-stage model approach. We trained two FCN models: the first for predicting if the puck was on the images, the second for predicting the puck location on the image. If the model predicted that the puck was on the image, the second model tried to find its pixel coordinates. The results were also underwhelming, most likely due to the multiplicity of uncertainty inherent in using two models for prediction.

During training we also experimented with various image transformations such a [redacted]

Controller Design Experiments

Our first controller operated on the knowledge of real-world coordinates. However, we could not retrieve these coordinate values directly from our model, so we modified our final implementation to instead [redacted]

In designing the controller, we also encountered several problematic kart behaviors that we needed to take into account. For one, our initial implementations would result in the kart getting stuck in its own goal, a rival's goal, or in front of a wall. As a result, we created logic for the kart to recognize this situation and [redacted]

Further, the kart also would attack other karts, the goal itself, and irrelevant objects like boxes or walls instead of going for the puck. The kart would also struggle to detect the puck if it was very distantly located in the image. However, a better model with a large training set was able to remedy these last two problems, and incorporating it into the controller ultimately produced kart behavior that recognized the puck in several different situations.

CONCLUSION

In this report, we document and evaluate the creation of an image-based agent designed for SuperTuxKart ice-hockey. The project's aim was to develop an autonomous agent capable of competing in 2 vs 2 matches within the SuperTuxKart ice-hockey framework with the goal of maximizing the number of points scored in a series of matches against various programmed agents.

Our method involved combining two vision-based algorithms for object recognition, in tandem with a manually adjusted controller for in-game navigation and strategy formulation. We created a novel data generation mechanism that involved the generation of a training image dataset from local gameplay against different AI opponents. It necessitated several important innovations to address problems in coordinate mapping, image sizing, and image transformation.

The most effective version of our implementation utilized a dual-model approach in the controller for maneuvering the kart, which relied on a pre-trained model to stag[redacted] a trained model to navigate the kart [redacted] in each match. Our implementation was successful. It scored several goals against various other grader agents.

Nonetheless, there are several areas of improvement in which, if more time and effort were invested, would result in better agent performance. The first relates to generating the training data. [redacted] also produce improved results.

There are also several improvements we could make to the controller. We noticed [redacted] This could be improved by training an even better model using the HW4 assignment code.

Further, we could implement logic in the controller to account for [redacted] This could produce a better overall controller.

While this SuperTuxKart ice-hockey game is just a small self-contained project, it underscores the importance of computer vision in similar tasks with real-world implications. The development of an image-based agent in the game serves as a model for understanding important computer vision challenges, mirroring key tasks in advanced applications like self-driving cars. Techniques used in object detection and spatial analysis within the game parallel those needed for autonomous vehicles to navigate and interpret their surroundings in real-world environments. Training the agent to react to its environment in a dynamic and flexible way offers insights into how autonomous systems can be taught to understand and react to complex, real-world situations. The success we achieved in our implementation not only demonstrates the effectiveness of current computer vision techniques but also highlights the potential for these technologies in shaping future advancements in fields like autonomous transportation.

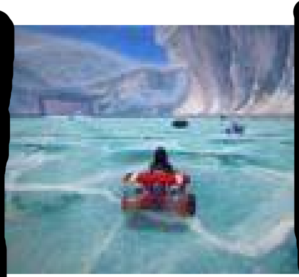
REFERENCES

1. Anwar, A. (2022, February 28). What are Intrinsic and Extrinsic Camera Parameters in Computer Vision?. Towards Data Science. https://towardsdatascience.com/what-are-intrinsic-and-extrinsic-camera-parameters-in-computer-vision-7071b72fb8
2. Ronneberger, O., Fischer, P., & BroxA, T. (n.d.). *U-Net: Convolutional Networks for Biomedical Image segmentation - arxiv.org*. arXiv. https://arxiv.org/pdf/1505.04597.pdf
3. O'Sullivan, C. (2023, March 8). *U-Net explained: Understanding its image segmentation architecture*. Medium. https://towardsdatascience.com/u-net-explained-understanding-its-image-segmentation-architecture-56e4842e313a

APPENDIX

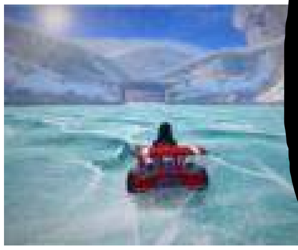
Appendix A: Additional Training Examples [redacted]

Images [redacted]

[redacted]

[redacted]

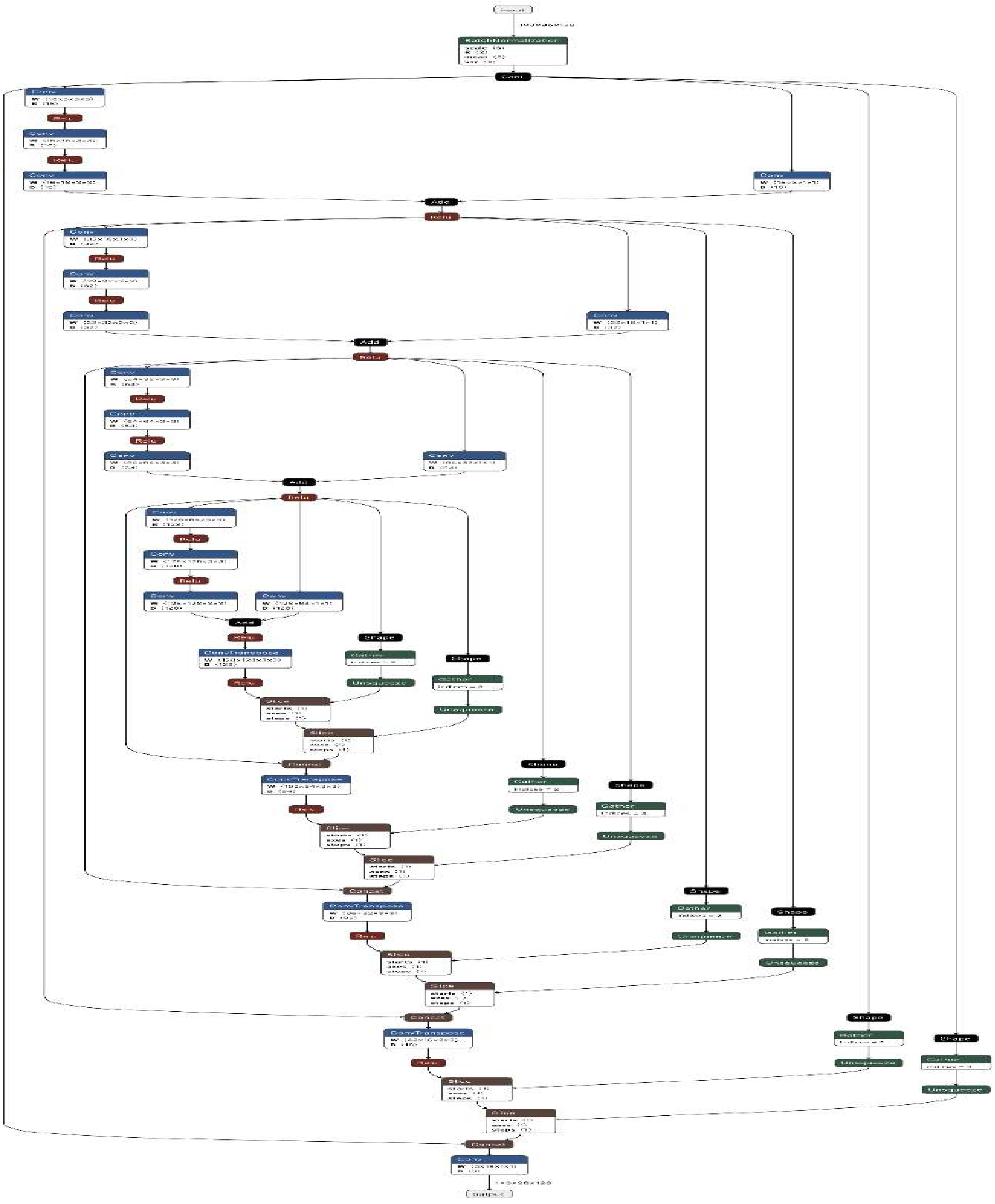
[redacted]

Images [redacted]


[redacted]

[redacted]

Appendix B: FCN Full Trained Model Architecture (generated by https://netron.app/)



Appendix C: Tabular View of Training and Validation Accuracy and Loss for 50 Training Epochs for Final Trained Model

epochs	Training Set		Validation Set	
	loss	accuracy	loss	accuracy
0	0.1253	0.8747	0.0513	0.9487
1	0.0451	0.9549	0.0415	0.9585
2	0.038	0.962	0.0365	0.9635
3	0.0336	0.9664	0.0333	0.9667
4	0.0311	0.9689	0.0312	0.9688
5	0.0294	0.9706	0.0295	0.9705
6	0.0282	0.9718	0.0295	0.9705
7	0.0277	0.9723	0.0288	0.9712
8	0.027	0.973	0.0274	0.9726
9	0.0259	0.9741	0.0282	0.9718
10	0.0262	0.9738	0.0264	0.9736
11	0.0251	0.9749	0.0255	0.9745
12	0.0246	0.9754	0.0259	0.9741
13	0.0243	0.9757	0.0248	0.9752
14	0.024	0.976	0.0252	0.9748
15	0.0241	0.9759	0.0255	0.9745
16	0.0237	0.9763	0.0244	0.9756
17	0.0232	0.9768	0.0239	0.9761
18	0.0231	0.9769	0.024	0.976
19	0.0228	0.9772	0.0238	0.9762
20	0.0229	0.9771	0.0234	0.9766
21	0.0228	0.9772	0.0234	0.9766
22	0.0228	0.9772	0.0231	0.9769
23	0.0224	0.9776	0.0229	0.9771
24	0.0224	0.9776	0.0227	0.9773
25	0.022	0.978	0.023	0.977
26	0.0219	0.9781	0.0228	0.9772
27	0.0222	0.9778	0.0228	0.9772
28	0.0218	0.9782	0.0226	0.9774
29	0.022	0.978	0.023	0.977
30	0.0222	0.9778	0.0222	0.9778

31	0.0217	0.9783	0.0229	0.9771
32	0.022	0.978	0.0221	0.9779
33	0.0215	0.9785	0.022	0.978
34	0.0216	0.9784	0.0221	0.9779
35	0.0214	0.9786	0.0222	0.9778
36	0.0216	0.9784	0.0216	0.9784
37	0.0218	0.9782	0.022	0.978
38	0.0218	0.9782	0.0223	0.9777
39	0.0214	0.9786	0.0222	0.9778
40	0.0212	0.9788	0.0224	0.9776
41	0.0214	0.9786	0.0216	0.9784
42	0.0209	0.9791	0.0212	0.9788
43	0.0213	0.9787	0.0216	0.9784
44	0.0209	0.9791	0.0214	0.9786
45	0.0213	0.9787	0.0217	0.9783
46	0.0211	0.9789	0.0216	0.9784
47	0.021	0.979	0.0217	0.9783
48	0.0211	0.9789	0.0214	0.9786
49	0.0206	0.9794	0.0213	0.9787